LOAD PARTITIONING FOR MATRIX-MATRIX MULTIPLICATION ON A CLUSTER OF CPU-GPU NODES USING THE DIVISIBLE LOAD PARADIGM

by

Lamees Elhiny

A Thesis Presented to the Faculty of the
American University of Sharjah
College of Engineering
in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science in
Computer Engineering

Sharjah, United Arab Emirates

November 2016

Approval Signatures

We, the undersigned, approve the Master's Thesis of Lamees Elhiny

Thesis Title: Load Partitioning for Matrix-Matrix Multiplication on a Cluster of CPU-GPU Nodes Using the Divisible Load Paradigm

Signature	**Date of Signature** (dd/mm/yyyy)
Dr. Gerassimos Barlas Professor, Department of Computer Science and Engineering Thesis Advisor	
Dr. Khaled El-Fakih Associate Professor, Department of Computer Science and Engineering Thesis Committee Member	
Dr. Naoufel Werghi Associate Professor, Electrical and Computer Engineering Department Khalifa University of Science, Technology & Research (KUSTAR) Thesis Committee Member	
Dr. Fadi Aloul Head, Department of Computer Science and Engineering	
Dr. Mohamed El-Tarhuni Associate Dean, College of Engineering	
Dr. Richard Schoephoerster Dean, College of Engineering	
Dr. Khaled Assaleh Interim Vice Provost for Research and Graduate Studies	

Acknowledgements

I would like to express my sincere gratitude to my thesis advisor, Dr. Gerassimos Barlas, for his guidance at every step throughout the thesis.

I am thankful to Dr. Assim Sagahyroon, and to the Department of Computer Science & Engineering as well as the American University of Sharjah for offering me the Graduate Teaching Assistantship, which allowed me to pursue my graduate studies.

Last but not least, I would like to thank my mother, Dr. Zeinab Demerdash, and my husband, Dr. Hany Taha, for their encouragement, love, and sacrifices. Without their efforts I would not have been able to carry out my work.

Dedication

To my late father, Dr. Mostafa Elhiny, and my late grandfather, the legal advisor Abu-Bakr Demerdash. Thank You . . .

Abstract

Matrix-matrix multiplication is a component of many numerical algorithms; however, it is a time consuming operation. Sometimes, when the matrix size is huge, the processing of the matrix-matrix multiplication on a single processor in not sufficiently fast. Finding an approach for efficient matrix-matrix multiplication can scale the performance of several applications that depend on it. The aim of this study is to improve the efficiency of matrix-matrix multiplication on a distributed network composed of heterogeneous nodes. Since load balancing between heterogeneous nodes forms the biggest challenge, the performance model is derived using the Divisible Load Theory (DLT). The proposed solution improves performance by: (a) the reduction of communication overhead, as DLT-derived load partitioning does not require synchronization between nodes during processing time, and (b) high utilization of resources, as both Control Processing Unit (CPU) and Graphical Processing Unit (GPU) are used in the computation. The experiments are conducted on a single node as well as a cluster of nodes. The results prove that the use of DLT equations balances the load between CPUs and GPUs. On a single node, the suggested hybrid approach has superior performance when compared to C Basic Linear Algebra Subroutines (cBLAS) and OpenMP Basic Linear Algebra Subroutines (openBLAS) approaches. On the other hand, the performance difference between the hybrid and GPU only (CUDA Basic Linear Algebra Subroutines) approaches is mild as the majority of the load in the hybrid approach is allocated to the GPU. On a cluster of nodes, the computation time is reduced to almost half of the GPU only processing time; however, the overall improvement is impeded by communication overhead. It is expected that faster communication media could reduce the overall time and further improve speedup.

Search Terms: ***hybrid processing, parallel processing, load partitioning, matrix-matrix multiplication, divisible load theory***

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

Currently, a common practice to achieve faster computations involves the composition of distributed systems by interconnecting commodity desktops. The potential for accelerated execution depends on workload divisibility among the connected nodes for parallel program processing. However, the success of parallel techniques is directly related to efficient load balancing. The work load should be partitioned and distributed in a way that reduces the program overall execution time [1]. Thus, all the time spent in communication, either to distribute data from the master node to the workers, or to collect the results from workers, should be reduced or overlap computation. The desired performance enhancement will not be achieved unless the proper workload balancing strategy is applied.

There are two methodologies used for load distribution, the static, and the dynamic approach. The static approach splits the data and assigns the partitions to processors before computation commences. On the other hand, the dynamic scheduling takes into consideration the real execution time of every partition and adjusts the load distribution schedule accordingly. Thus, the dynamic strategy can adapt to changes during runtime such as network traffic and off-line nodes. However, the major disadvantage of this approach is the overhead caused by extra communication and frequent synchronization between nodes. In certain situations, the overhead caused by the use of the dynamic workload balancing can degrade the overall performance [2, 3].

A large number of load balancing algorithms is based on a theory called Divisible Load Theory (DLT); DLT popularity is obtained from its ability to define a mathematical model used to do time-optimal processing. DLT assumes that the data can be divided into arbitrary, independent partitions that can be processed in parallel. These partitions should not have any precedence relations. This requirement is satisfied by a wide spectrum of scientific problems such as digital image processing, database processes and linear algebra calculations [3–9]

Most machines nowadays are equipped with a Graphical Processing Unit (GPU) besides the Central Processing Unit (CPU). A CPU contains few cores optimized for sequential processing while a GPU has a massively parallel architecture composed

of thousands of smaller, simpler cores designed for executing multiple tasks simultaneously. As its name might suggest, GPUs were first used for manipulating computer graphics. Later, General Purpose GPU Processing (GPGPU), in which compute-intensive portions of the application are off-loaded to the GPU, has evolved. GPGPU is now used to accelerate scientific, analytics, engineering, consumer, and enterprise applications [10].

The speedup achieved by GPUs, when used in certain applications, made them a crucial component in parallel architecture. With the evolution of multi-core CPUs, developers started to make use of the extra cores through designing applications that can be executed in parallel. Recently, hybrid computation started to grab attention: instead of using multi-cores alone or GPU alone, why not integrate both to optimize the use of available resources? The use of GPU and CPU as peers can boost performance particularly in data-intensive applications, applications that process huge data sets. However, the simultaneous use of GPU and CPU requires challenging scheduling techniques, which should take into account the difference in computation costs and capabilities between these two hardware platforms.

The interest in hybrid parallel computation has grown considerably in recent times. The scheduling techniques used are tightly coupled with the application [11]. Several algorithms were proposed to tackle the load balancing issue of linear algebra operations, including matrix multiplication, in hybrid systems. In addition, some linear algebra libraries like cuBLAS-XT provided routines that allow the simultaneous use of CPU and GPU. However, all these methodologies failed to provide a mathematical tool that can be utilized to handle workload distribution in heterogeneous systems.

This thesis suggests the use of DLT to provide a mathematical model that will be used for load balancing between the CPU and GPU during matrix multiplication. DLT mathematical models can also be used to measure if the node's contribution will actually enhance performance or not. Further analysis based on load distribution equations can reveal the minimum and maximum number of nodes that can participate in a network in order to reduce the overall execution time.

The contribution of this thesis can be summarized as follows:

- Providing a DLT solution for matix-matrix multiplication, one of the most expen-

sive basic linear algebra routines

- Offering a performance evaluation of the proposed mathematical model

The organization of this thesis is as follows. Chapter 2 includes the problem definition and approach. Chapter 3 includes the literature review. Chapter 4 discusses the performance model as well as the DLT equations. Chapter 5 covers the experiment set-up as well as the measurement of parameters. Chapter 6 contains the implementation and profiling details. In Chapter 7, the experimental evaluation of the conducted experiments is discussed and Chapter 8 concludes this thesis.

Chapter 2: Problem Definition and Approach

Matrix-matrix multiplication is a time consuming operation, that is a component of many numerical algorithms. When the matrix size is huge, matrix-matrix multiplication on a single processor is tremendously slow. Using parallel processing for speeding-up matrix-matrix multiplication can enhance the performance of several applications.

In this thesis, we consider a group of heterogeneous nodes such that each node is equipped with multi-core CPU and a GPU. Our target is to accelerate large sized matrix-matrix multiplication through high resource utilization. Thus, the work load is first divided between the nodes in the network. Then, the distributed partitions is subdivided between the processors in a single node. To achieve the optimum performance, two distinct scheduling problems must be addressed: inter node and intra node load balancing.

2.1. Inter Node Load Balancing

A master/slave paradigm is used in inter node load balancing. One node hosts the matrices; assuming that the computational capability of this node is insufficient for matrix multiplication optimal processing, the matrices are partitioned using the divisible load methodology, and distributed to other nodes for parallel processing using the Message Passing Interface (MPI). The introduced approach enables communication/computation overlap to hide communication cost.

2.2. Intra Node Load Balancing

In intra node load balancing, the work is subdivided among node processors. The node available CPUs and GPUs are queried and, according to their number as well as their computational power, the work load is distributed among them to achieve the most efficient load processing. For example, if a node has four cores and a GPU, three cores will participate in computation, while the fourth one will be used to copy the data to/from the GPU. The Compute Unified Device Architecture platform (CUDA) is used to program the GPU. Asynchronous CUDA APIs are used whenever possible to reduce

communication overhead resulting from copying huge matrices from host to device and vise versa.

2.3. GPU Programming Characteristics

The two major factors that greatly affect GPU use in parallel processing are:

- Input Size: The size of the input plays a crucial role in determining the most efficient hardware set up for task execution. It should not be taken for granted that the use of GPU will necessarily enhance performance. On the contrary, it can drastically increase the total execution time. Copying the data from host to device and vice versa is an expensive process; hence, GPU performance enhancement can only be achieved when the size of data is big enough such that the reduction in computation time can offset the communication overhead [10]. For every application, there is a certain size threshold below which GPU execution deteriorates performance.
- CUDA Runtime Initialization: There is a delay that occurs when the first runtime CUDA call, usually cudaMalloc(), is made. This delay is caused by CUDA runtime initialization. This CUDA startup time must be taken in consideration whenever a GPU is to be used in parallel processing [12].

Chapter 3: Related Work

The literature review is divided into two parts, the first one covers research work done in the area of DLT and parallel computations using heterogeneous clusters. The second part of this section lists different linear algebra libraries that include matrix-matrix multiplication aiming at discussing the pros and cons of each of these libraries.

3.1. Research Work

This section is divided into two parts. The first part discusses heuristic algorithms used to solve a number of linear algebra problems using parallel processing, while the second one covers DLT load partitioning techniques.

3.1.1. Heuristic algorithms. Park and Perumalla state in [13] that efficient use of hybrid systems in linear algebra computations can reduce total execution time. Despite the speed-up achieved by using GPU parallel processing for solving linear algebra problems, the heterogeneous parallel systems can still compete and further enhance performance. The authors argue that in GPU only parallel set-up, the CPU computation capabilities are wasted by being restricted to inter-node communication (data supply to the GPU). On the other hand, their study shows that blind use of GPU/CPU parallel structure would not produce the desired effect unless applied appropriately. To derive benefit from hybrid parallel execution, proper memory management of the GPU (in case of several processes accessing the accelerator simultaneously) as well as efficient load distribution between GPU and CPU should be taken into consideration.

One of Park and Perumalla's contributions to optimize hybrid computation is the libaccelmm library which handles GPU memory management [13]. This library can be utilized by applications that reuse computational results performed by the accelerator. Libaccelmm treats the GPU memory as a virtual memory and the CPU memory as a disk. The only limitation is that the data required for a given process computation must completely fit in the GPU memory. The role of libaccelmm library can be summarized as follows:

- Device memory mapping: This includes handling data copying from host to device, and device memory allocation. In addition, a new entry is created in a hash table to keep track of the copied data. The hash table entry has two pointers and a number of flags. The pointers are pointing to data storage in the host and to its location in the device. The flags are required for data validation and efficient memory management.
- Device memory synchronization: When the host data or the device data are modified, the corresponding hash table entry is marked as invalid. In this case, libaccelmm is responsible for data synchronization between the device and the host before being accessed by any process.
- Device memory deletion and replacement: libaccelmm ensures that deleting data will not affect correctness but can affect performance. The library keeps track of the least and the most recently used data, and whenever the device is out of memory, the least recently used data are evacuated first.

Park and Perumalla tested their implementations by solving a system of linear equations:

$$Ax = b \tag{1}$$

where A is a $NM \times NM$ block tridiagonal matrix (A is described as a $N \times N$ matrix of $M \times M$ blocks), x and b are $NM \times 1$ vectors (formed by writing the columns of $N \times M$ matrix one below the other) [14]. The mathematical operations required to solve the problem involves the following four steps (performed on blocks) :

1. A factorization of the $M \times M$ diagonal block matrix
2. Two solve operations on $M \times M$ using the factorization calculated in step 1
3. Two matrix-matrix multiplications
4. A matrix-vector multiplication

Park and Perumalla used a cyclic reduction algorithm, as well as divide and conquer strategies to tackle the problem [15]. Park and Perumalla experiment set-up was as follows:

- Software set-up: libaccelmm is written in CUDA. Solving the tridiagonal matrix required BLAS and LAPACK routines in CPU execution and cuBLAS and MAGMA in GPU execution.

- Hardware set-up: TitanDev was the platform used; it is a supercomputer containing 15,360 cores among 960 nodes, each consisting of one 16-core AMD Interlagos processor with 32 GB of memory and one nVidia TX2090 accelerator connected via PCI Express.

Park and Perumalla compared four different execution structures. CPU only execution in which multi-cores are involved. GPU only execution, in which the role of the CPU is to supply data to the GPU. CPU/GPU structure in which one process is used for CPU parallel execution and another for GPU. SGPU/CPU in which multiple processes are allowed to access the GPU. In the SGPU/CPU scenario proper partitioning is addressed plus efficient memory management (using libaccelmm).

In [16], Ravi et al. propose a number of new scheduling algorithms that enable load balancing on CPU-GPU clusters. Their proposed solution is based on the fact that different tasks have different performances on different resources. Thus, tasks should be assigned to the resource that will execute them faster, and not in a first come first served scheme. The authors assume that the program can be divided into tasks compatible to run on both the CPU or the GPU. The implemented scheduler decides the best set up to execute the task whether it is the CPU cores, the GPU or both, in any number of nodes in the network. The obtained results show that the suggested load balancing scheme outperforms a blind round-robin (naive dynamic approach) methodology and approaches the performance of an ideal scheduler that includes an idealistic exhaustive investigation of all possible schedules.

A recent study was done by Zhu et al. [17] targeting heterogeneous computation support. In hybrid systems, there is a necessity for code compatibility between CPUs and GPUs as well as different kinds of GPUs [16, 17]. A task can be mapped to either GPU or CPU; however, tasks must be distributed based on performance. Not all tasks perform well on GPU, only hot spots of the code (such as loops with no data dependencies) can benefit from GPU parallel computation. The authors achieved code compatibility between CPU and GPU using a dynamic binary translator called CrossBit. CrossBit first translates binary source code to an intermediate instruction set and then transforms these instructions to the target platform code. The researchers developed a module GXBIT which employed the CrossBit to support hybrid computation as

follows [17].

- CrossBit is used to convert binary source code to intermediate instruction set
- The hot spots of the program are extracted and the required information is gathered
- The code is then translated to relevant platform code (CUDA for hot spots of the code)

In [18], Lastovetsky and Reddy suggest a load balancing algorithm for some computations including matrix-matrix multiplication that considers memory constraints of the processors. Primarily, the algorithm calculate the partition for the processors based on their computational capabilities. Afterwards, the calculated partition for each processor is compared with the size of its memory. In case the allocated partition does not fit in the processor memory, the part assigned to that processor is reduced to the maximum size that fits while the rest is redistributed between the remaining nodes.

3.1.2. DLT methodologies. As mentioned before, DLT requires that the workload can be divided into several independent arbitrary sized chunks that can be processed in parallel. A summary of application domains that satisfy this criterion is listed in Table 1. This is followed by detailed explanation of some DLT applied solutions.

Barlas, Hassan and Al Jundi [19] state that there is a necessity to take advantage of both CPU cores and GPU devices to speed-up tasks. In [19], Barlas, Hassan and Al Jundi discuss the use of both GPU and CPU to fasten the encryption and decryption process of block ciphers. The study has the following contributions:

- A mathematical framework based on DLT is proposed to optimally distribute/collect data to/from hybrid nodes. This is explained in details below.
- The proposed partitioning approach showed better results than the dynamic load balancing one.

Barlas, Hassan and Al Jundi's architecture is composed of N heterogeneous worker nodes receiving input data from one node named the Coordinator Node (CN). In their proposed model, the CN is not contributing in the computations but with slight adjustments to cost parameters it can. The total processing cost (T_i) of a portion ($part_i$) of the block ciphers can be precisely modeled as the summation of the distribution

Table 1: List of applications for which closed form partitioning solutions have been obtained

Applications	Authors	Reference(s)
Video Compression	Barlas, Li, Veeravalli, Kassim, Momcilovic, Illic, Roma, Sousa	[20–22]
Cloud System	Suresh, Huang, Kim, Abdullah, Othman	[23]
Image Processing	Lee, Hamdi, Veeravalli, Li, Ko, Ranganath	[24–27]
Multiple Protein Sequence Alignment	Low, Veeravalli, Bader	[28]
Biological Sequence (DNA) Comparison	Min, Veeravalli	[29]
Wireless Sensor Networks	Shi, Wang, Kwok, Chen,,Moges, Robertazzi	[30–33]
Resilient Lambda Grids	Thysebaert, De Leenheer, Volckaert, De Turck, Dhoedt, Demeester	[34]
Data Grid Applications	Abdullah, Othman, Ibrahim, Subramaniam	[35]

(DS_i), the collection (CL_i) and the processing cost (PR_i) [19]:

$$PR_i = p_i part_i L \tag{2}$$

$$DS_i = l_i(part_i L + k) + a_i \tag{3}$$

$$CL_i = l_i part_i L + a_i \tag{4}$$

The symbols used above are explained in Table 2. Assuming parallel input distribution, the minimum time to process L can be achieved when all the nodes begin and end computation at the same time. Thus, for two nodes i and j where i, j in $[0, N-1]$

$$part_i = part_j \frac{p_j + 2l_j}{p_i + 2l_i} + \frac{2(a_j - a_i) + k(l_j - l_i)}{L(p_i + 2l_i)} \tag{5}$$

Table 2: Symbol table for [19].

Symbol	Description
L	Data to be encrypted/decrypted
$part_i$	Portion of the data $0 <= part_i <= 1$
p_i	Inversely proportional to processor speed
k	Size of encryption key
a_i	Communication latency
l_i	Inverse of node i's communication link data rate
M	The total number of installments in case communication capabilities of a node exceeds its computation

$$\sum_{i=0}^{N-1} part_i = 1 \tag{6}$$

Using equations (5) and (6) $part_0$ can be calculated as follows:

$$part_0 = \frac{1 - \sum_{i=1}^{N-1} \frac{2(a_0 - a_i) + k(l_0 - l_i)}{L(p_i + 2l_i)}}{1 + \sum_{i=1}^{N-1} \frac{p_0 + 2l_0}{p_i + 2l_i}} \tag{7}$$

From equations (5) and (7), we notice that for a certain node $part_i$ can be negative. This indicates a slow node. One way to deal with the slow nodes is to remove them from computation.

In case that communication time exceeds processing time, the total processing cost can be reduced by subdividing $part_i$. $part_i$ can be supplied to the processors as installments. In that way, the communication and the processing can overlap and the $l_i k$ overhead in (2) will apply only for the first installment. In the multi-installment case T_i can be calculated as follows [19]:

$$T_i = l_i(part_{0,i}L + part_{M-1,i}L + k) + 2a_i + p_i L \sum_{j=0}^{M-1} part_{j,i} \tag{8}$$

In [5], Ilic and Sousa state that little effort was spent on the study of DLT in highly heterogeneous systems in which the computation is distributed among computer devices as well as CPUs and GPUs available in each device. The researchers propose a feasible solution to model the relative performance of system resources that are not known in advance. Their algorithm adopts an iterative procedure that is composed of two main phases: initialization and the iterative phase. The initialization phase is responsible for the determination of α (partition offloaded to a single machine), β (partition supplied to each processor either a CPU or GPU in one machine) and preliminary γ (installment given to GPU) partitions. The iteration phase commences with the continuous splitting of γ into sub loads using a factor by two technique to achieve the optimum load balancing. After each γ split, the new α and β are computed and the performance is assessed. If there is no significant improvement in performance from the previous run, the current α, β, and γ partitions are considered the most efficient values and load balancing is achieved; otherwise, the iteration is repeated. The above notations as well

Table 3: A summary of notation for [5].

Symbol	Description
N	The whole data
α	Portion of data assigned to different computer devices in the network (inter- node partitions)
β	Portion of data split among different processors (CPUs and GPUs of the same device i.e. intra-node partitions)
γ	The installment supplied to the GPU
D	The desktops in the network
$\Psi_{\mathscr{T}}(x)$	The total time to distribute and process load of size x on a desktop system (total relative performance for the desktop)
m	The number of CPU cores in a single desktop
w	The number of devices like GPU in a single desktop
$\psi_{\tau}(x)$	The ratio between the load of size x and the time required to communicate and process x in a single processor in D_i (total relative performance for the processor where i is the desktop index)
Γ	The total number of sub-loads (resulting from $\beta_{i,j}$ subdivision where i is the desktop index, and j is the processor index)
Γ^k	The total number of sub-load fractions (resulting from γ^k sub-partitioning), where k is $1 <= k <= \Gamma$

as other symbols used in Ilic and Sousa's DLT solution are explained in Table 3.

The authors proposed algorithm for load scheduling is composed of the three main steps [5] :

Step 1- Calculating α

$$\frac{\alpha_1}{\Psi_{\tau_1}(\alpha_1)} = ... = \frac{\alpha_{|D|}}{\Psi_{\tau_{|D|}}(\alpha_{|D|})}; \sum_{i=1}^{|D|} \alpha_i = N \tag{9}$$

Step 2- Calculating β

$$\frac{\beta_{i,1}}{\psi_{\tau_{i,1}}(\beta_{i,1})} = ... = \frac{\beta_{i,m+w}}{\psi_{\tau_{i,m+w}}(\beta_{i,m+w})}; \sum_{j=1}^{m+w} \beta_{i,j} = \alpha_i \tag{10}$$

Step 3- Calculating γ. In distant workers, the load is sub-partitioned into γ to allow communication and computation overlap

$$\sum_{k=1}^{|\Gamma|} \sum_{l=1}^{|\Gamma^{(k)}|} \gamma_l^k = \beta_{i,j} \tag{11}$$

Note that the sum of sub-fractions (γ_l^k) should fit in the GPU memory.

The most relevant study to this thesis is the one conducted by Chan, Bharadwaj, and Ghose [36] on large matrix-vector multiplication using DLT. The researchers used several identical processors linked through a bus network as shown in Figure 3. The load was bigger than the computation capability of a single node (the master node). Consequently, the node divided the workload and distributed it to be processed using several machines then collected the results. The master node did not participate in the computation. Similar to this thesis topic, the researchers tried to find the ultimate speed-up using DLT analysis. Despite the communication delay, the authors were able to achieve a closed form solution to the problem which they further used to determine the minimum and maximum number of nodes that can share in processing and enhance performance [36]. Chan, Bharadwaj, and Ghose study divides the matrix row-wise on a group of homogeneous nodes. On the other hand, this thesis targets highly heterogeneous networks inwhich both inter and intra node load balancing should be achieved.

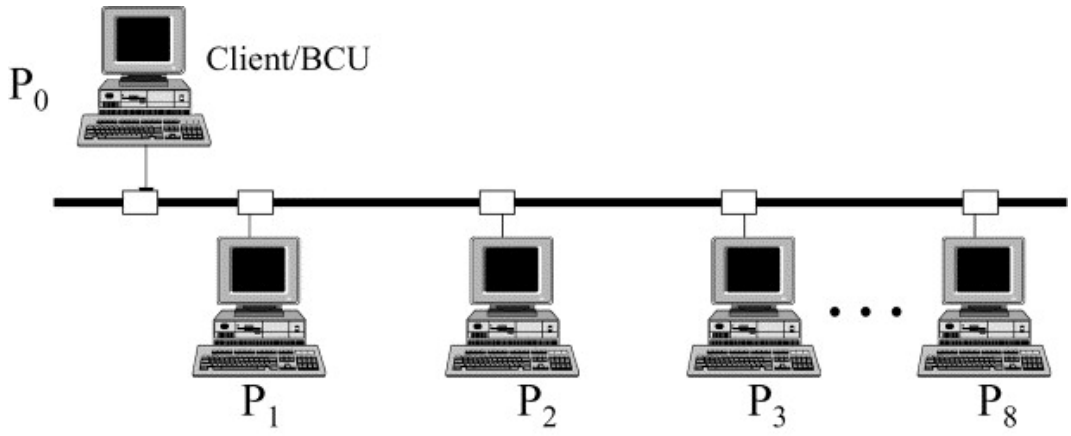

Figure 1: Distributed network used in matrix-vector multiplication [36].

3.2. Linear Algebra Libraries

Due to the importance of basic linear algebra routines in solving a huge number of complicated real life problems, a large number of optimized libraries have been developed. The earlier versions of such libraries adopted sequential algorithms. With the evolution of parallel programming, parallel versions of these routines were released.

GPUs were the core accelerator of matrices operations. Recently, some libraries targeted systems in which load can be dispatched between CPUs and GPUs.

3.2.1. Basic linear algebra subprogram. Basic Linear Algebra Subprogram (BLAS) is an open source library that provides a set of optimized basic matrix operations. Due to its efficiency, availability, and portability, the BLAS is used in several linear algebra applications like Linear Algebra Package (LAPACK[1]). LAPACK is a software for solving complicated linear algebra problems such as a system of simultaneous linear equations and eigenvalue problems. BLAS was orginally written in Fortran; however, a C version of BLAS was released later [37]. Furthermore, parallel implementations of BLAS evolved such as openBLAS[2], clBLAS and cuBLAS.

BLAS is composed of three modules:

- BLAS 1: contains routines to perform vector-vector operations
- BLAS 2: responsible for vector-matrix operations
- BLAS 3: methods that perform matrix-matrix operations [37]

3.2.2. Linear algebra libraries supporting hybrid processing. A summary of linear algebra libraries supporting hybrid computation is given in Table 4. A detailed description of these libraries is given below.

3.2.2.1. CuBLAS-XT. CuBLAS was implemented using CUDA to run on GPUs. Currently, the latest version contains a group of subroutines called cuBLAS-XT; these modules allow hybrid CPU-GPU computation. CublasXT version supports CPU-GPU load distribution strategy using two routines: cublasXtSetCpuRoutine() and cublasXtSetCpuRatio(). These functions can be used together to setup the percentage of load that will be assigned to the CPU. These functions are only supported for xGEMM routines (Matrix-Matrix operations) [38].

3.2.2.2. AMD Core Math Library. AMD Core Math Library (ACML) provides a free set of efficient threaded math routines. ACML contains a full implementation of BLAS Level 1, 2 and 3, with key modules optimized for high performance on

[1] http://www.netlib.org/lapack/
[2] http://www.openblas.net/

AMD Opteron^{TM} processors. ACML 6 permits the heterogeneous computation of all BLAS Level 3 subroutines and two Level 2 subroutines (GEMV & SYMV) in which the load can be divided between CPUs and GPUs. The GPU processing is achieved by calling the clBLAS library that ships with ACML6. The load partitioning heuristic logic is controlled by ACML scripting language. The ACMLScript is a scripting language embedded in the ACML library, introduced with version 6. It allows ACML to embed programming logic within text files, which avoids hard-coding logic within the library itself. The main purpose of ACMLScript is to encode the load balancing logic in scripts. This allows the logic to be updated to fit the needs of the user [39].

3.2.2.3. Intel Math Kernel Library. Intel Math Kernel Library (Intel MKL) is a mathematical library that provides a set of threaded routines including linear algebra operations. Natively, Intel MKL supports C, C++ and Fortran development. A recent addition is the support of an OpenCL SDK which enables GPU-CPU parallel processing. The success of using the OpenCL SDK is case dependent. The speed-up achieved from a hybrid execution depends on application characteristics such as the fraction of parallel work, data dependencies, and synchronization requirements [40, 41].

3.2.2.4. HiFlow. HiFlow is multi-purpose software that provides powerful tools for efficient and accurate solutions of a wide range of medical and industrial problems modeled by partial differential equations (PDEs). The goal of HiFlow is the full utilization of resources in hybrid platforms ranging from supercomputers to stand-alone desktops. To achieve this goal, HiFlow uses MPI to manage communication between processors as well as a hardware-aware computing modules implemented on the linear algebra level. The target of HiFlow is to supply the user with methodologies and modules that can apply to a variety of problem classes and architectures [42].

Table 4: Summary of linear algebra libraries supporting hybrid computation

Libraries	**Supported Computing Devices**	**Routines Supporting Hybrid Feature**	**Load Partitioning Methodology**	**Restrictions**
CuBLAS-XT [38]	Single nVidia GPUs and dual-GPU cards such as the Tesla K10 or GeForce GTX690	xGEMM (BLAS LEVEL 3 matrix-matrix multiplication)	The user has to set-up the amount of work offloaded to the CPU	The user should be careful when using this feature as it could interfere with the CPU threads feeding the GPUs
ACML Version 6 [39]	AMD Opteron processors	All BLAS Level 3 subroutines and two Level 2 subroutines (GEMV & SYMV)	ACML uses heuristic algorithm to split data between CPU and GPU; it saves the load balancing logic in a text file	Does not support multiple GPU processing in a node
Intel MKL [40]	IntelHD Graphics devices	All implemented routines through the use of Intel OpenCL SDK	The load-balancing between CPUs and GPUs should be implemented	The success of the heterogeneous strategy should be studied for each application
HiFlow [42]	Intel and AMD multi-core CPUs and NVIDIA GPUs	BLAS 1 and 2 routines	Implemented modules handle load partitioning process based on available hardware	Only BLAS 1 and 2 are available

Chapter 4: Performance Model

This chapter starts with a detailed description of the performance model followed by the closed form solution. The closed form solution section contains a table of notations and the DLT equations as well as their explanation.

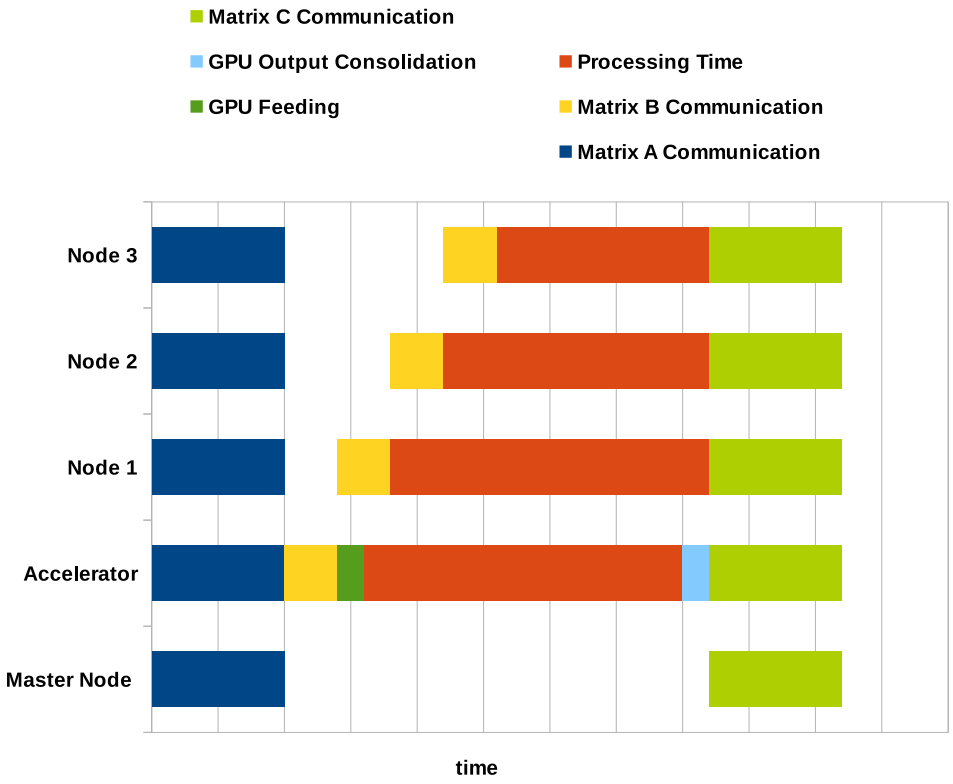

Figure 2: The communication and computation of matrices A and B on a hybrid system composed of an accelerator and three nodes having the same computational power.

4.1. Model Description

Suppose there are two matrices A of size $N \times M$ and B of size $M \times K$ where $K > N$. The matrices A and B are stored on one machine, the master node. This machine will be responsible for distributing data to worker nodes. The hardware set-up of the worker is composed of one GPU and three nodes (CPUs) having different computational capabilities. Assuming that the communication time for all the nodes is the same, load distribution of the two matrices for matrix multiplication can be described as follows:

- Matrix A will be distributed in parallel to all nodes.
- Matrix B is partitioned in column-wise fashion to uneven sized partitions and distributed to working nodes sequentially.
- The fastest node (the GPU) takes the first part of the load; notice that there will be extra communication overhead for the GPU.
- All nodes have to finish computation at the same time as shown in Figure 2. Note that the execution time of a task on a set of nodes is minimized if they all finish processing at the same time [43].

4.2. DLT Equations

This section contains the closed form partitioning solution for the matrix-matrix multiplication. The notations used are shown in Table 5.

Table 5: Symbols and Notations

Symbol	**Description**
$part_i$	Portion of the data $0 <= part_i <= 1$
p_j	Inversely proportional to processoring speed of the CPU
e_j	CPU processing latency
$p_{j,a}$	Inversely proportional to processoring speed of the GPU
$e_{j,a}$	GPU latency
l_p	Inverse of the GPU PCIe bus speed
l	Inverse of communication link data rate
b	Communication latency
s	The size of bytes of the data type used to represent matrix elements

4.2.1. The case of an accelerator, CPU-core pair. Suppose we have to multiply two matrices, A ($N \times M$) and B ($M \times K$). We assume that each multicore node (MN) in the system is assigned a number of columns of the result matrix C. Computation at a node will start after the whole of A is transmitted and the corresponding columns of B are also received. Computation at a core can commence after the columns of B corresponding to its own part of the C matrix are collected.

The time required for downloading A is

$$t_A = l\ N\ M\ s + b \tag{12}$$

where s is the size in bytes of the type used to represent matrix elements.

We can assume that t_A is a cost incurred by all MNs, except from the "load originating" one. We can also assume that it takes place in the form of a broadcast. The time required for downloading the c_j columns of B required by a core j is

$$t_{B_j} = l\ M\ c_j\ s + b \tag{13}$$

where $c_j = part_j K$. The computational cost for core j is:

$$t_j = p_j\ N\ M\ c_j + e_j \tag{14}$$

where e_j corresponds to constant setup overheads. These can be zero for CPUs or more significant for GPUs (e.g. the time required to initialize the CUDA runtime).

We first examine the case of a MN with an accelerator card and a single core. Accelerator cards (e.g. GPUs) require extra communication time over the PCIe bus for delivering the input data and retrieving the results. On the other hand, a CPU core can start computation as soon as the data are received by a MN.

The two possible timings are shown in Figure 3. Matrix A is delivered to the accelerator as soon as it is received by the MN system. Typically, the PCIe speed far exceeds the network speed, so we can assume with a degree of confidence, that A can cross the PCIe bus before the next portion of matrix B is delivered.

The cost of sending data over the PCIe bus is a linear function of the data volume (no latency used in this case):

$$t_{PCIe} = l_P\ s\ M\ K\ part_0 \tag{15}$$

The B matrix is divided between the two nodes shown in Figure 3 (i.e. $part_0 + part_1 = 1$), we have for the first configuration:

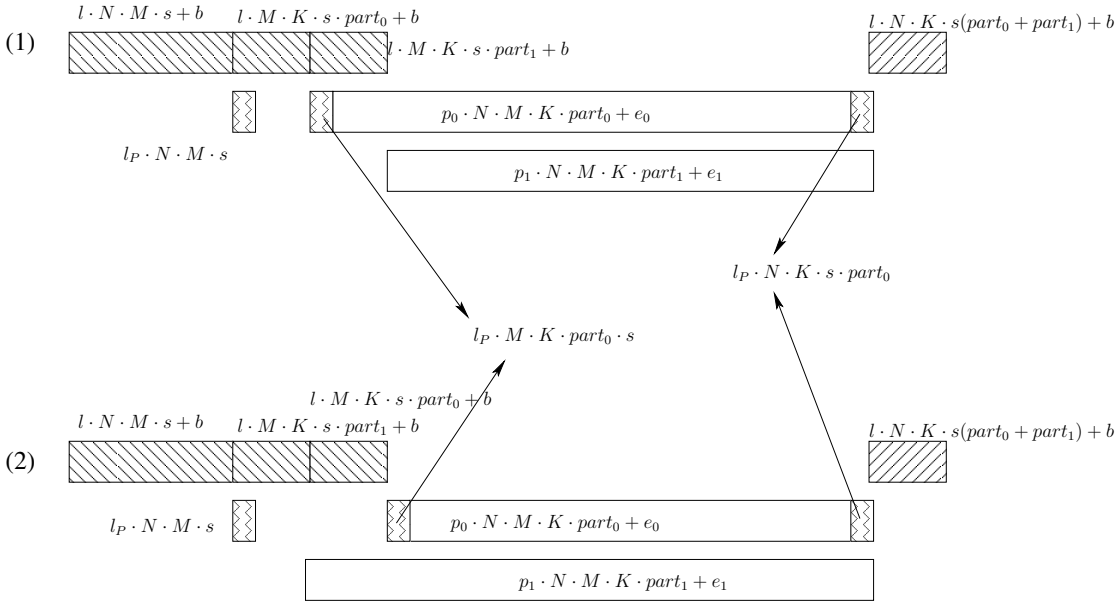

Figure 3: Possible timings for a CPU core and an accelerator card in processing the product of two matrices $A \times B$.

$$l_P MKspart_0 + p_0 NMKpart_0 + e_0 + l_P NKspart_0 = lMsKpart_1 + b + p_1 NMKpart_1 + e_1 \quad (16)$$

with the total time excluding receiving A and sending back C being equal to:

$$t_{(1)} = lMKs + 2b + p_1 NMKpart_1 + e_1 \quad (17)$$

For the second configuration we have:

$$p_1 NMKpart_1 + e_1 = (l + l_P)MKspart_0 + b + p_0 NMKpart_0 + l_P NKspart_0 + e_0 \quad (18)$$

with the total time excluding receiving A and sending back C being equal to:

$$t_{(2)} = lMKspart_1 + b + p_1 NMKpart_1 + e_1 \quad (19)$$

The parts can be found with the normalization equation and the difference between $t_{(1)}$

and $t_{(2)}$ can be simplified to:

$$t_{(1)} - t_{(2)} = \frac{((bp_0 - bp_1)M + bl_p s)N + ((e_1 - e_0)l + bl_p)sM}{((p_0 + p_1)M + l_P s)N + (l_P + l)sM} = \frac{bNM}{((p_0 + p_1)M + l_P s)N + (l_P + l)sM}(p_0 - p_1 + \frac{(e_1 - e_0)ls}{bN} + \frac{l_P s}{M} + \frac{l_P s}{N}) \quad (20)$$

The result obtained by equation (20) reflects the fact that the optimal ordering depends on both the relative speeds of the two computing platforms, but also on the severity of the constant overheads associated with the initialization of the computation.

4.2.1.1. Multiple single-accelerator-equipped systems. Assuming that the accelerators will receive their part of the B matrix first, we can establish relationships connecting the part to be assigned to the accelerator of the j-th system $part_{j,0}$ and the parts to be assigned to the remaining of its n_j cores $part_{j,i}$ for $i \in [1, n_j]$:

$$lsMKpart_{j,0} + b + l_P sMKpart_{j,0} + p_{j,a}NMKpart_{j,0} + e_{j,a} + l_P NKspart_{j,0} = lsMK\sum_{m=0}^{i} part_{j,m} + (i+1)b + p_j NMKpart_{j,i} + e_j \quad (21)$$

For $i = 1$ in the above equation we get:

$$lsMKpart_{j,0} + b + l_P sMKpart_{j,0} + p_{j,a}NMKpart_{j,0} + e_{j,a} + l_P NKspart_{j,0} = lsMK(part_{j,0} + part_{j,1}) + 2b + p_j NMKpart_{j,1} + e_j \Rightarrow part_{j,1} = part_{j,0}\frac{l_P s(1 + \frac{N}{M}) + p_{j,a}N}{ls + p_j N} + \frac{e_{j,a} - e_j - b}{MK(ls + p_j N)} \quad (22)$$

For two successive CPU cores we have:

$$p_j NMKpart_{j,i} + e_j = lsMKpart_{j,i+1} + b + p_j NMKpart_{j,i+1} + e_j \Rightarrow part_{j,i+1} = part_{j,i}\frac{p_j N}{ls + p_j N} - \frac{b}{MK(ls + p_j N)} \quad (23)$$

We can rewrite equations (22) and (23) as:

$$part_{j,1} = part_{j,0}Z_j + \Phi_j \quad (24)$$

$$part_{j,i+1} = part_{j,i}X_j + Y_j \tag{25}$$

with $Z_j = \frac{l_{PS}(1+\frac{N}{M})+p_{j,a}N}{ls+p_jN}$, $\Phi_j = \frac{e_{j,a}-e_j-b}{MK(ls+p_jN)}$, $X_j = \frac{p_jN}{ls+p_jN}$ and $Y_j = -\frac{b}{MK(ls+p_jN)}$ constants which are problem and platform specific.

We can also extend equation (25) to:

$$\begin{aligned} part_{j,i+1} = part_{j,i}X_j + Y_j = part_{j,i-1}X_j^2 + X_jY_j + Y_j = \ldots \\ = part_{j,1}X_j^i + Y_j\sum_{m=0}^{i-1} X_j^m = part_{j,1}X_j^i + Y_j\frac{X_j^i - 1}{X_j - 1} = \\ = part_{j,0}Z_jX_j^i + \Phi_jX_j^i + Y_j\frac{X_j^i - 1}{X_j - 1} \end{aligned} \tag{26}$$

An association between the parts assigned to two individual j and q MNs can be also established, by equating the total duration of their executions. The total execution time of a MN j is:

$$\begin{aligned} T_j = lsNM + b + lsMKpart_{j,0} + b + l_{PS}MKpart_{j,0} + p_{j,a}NMKpart_{j,0} + e_{j,a} + \\ l_{PS}NKpart_{j,0} + lNKs\sum_{m=0}^{n_j} part_{j,m} + b = \\ lsNM + 3b + part_{j,0}\left(lsMK + l_{PS}MK + p_{j,a}NMK + l_{PS}NK\right) + e_{j,a} + \\ lNKs\sum_{m=0}^{n_j} part_{j,m} \end{aligned} \tag{27}$$

The summation term in the left hand side of the above expression can be reduced using the following equations:

$$\sum_{m=1}^{n_r} X_r^{m-1} = \sum_{m=0}^{n_r-1} X_r^m = \frac{X_r^{n_r} - 1}{X_r - 1} \tag{28}$$

and

$$\begin{aligned} \sum_{m=1}^{n_r} \frac{X_r^{m-1} - 1}{X_r - 1} = \frac{1}{X_r - 1}(\sum_{m=1}^{n_r} X_r^{m-1} - \sum_{m=1}^{n_r} 1) = \\ \frac{1}{X_r - 1}(\sum_{m=0}^{n_r-1} X_r^m - n_r) = \frac{1}{X_r - 1}(\frac{X_r^{n_r} - 1}{X_r - 1} - n_r) \end{aligned} \tag{29}$$

The outcome of the above simplification is as follows:

$$\sum_{m=0}^{n_j} part_{j,m} =$$

$$part_{j,0} + part_{j,0} Z_j \sum_{m=1}^{n_j} X_j^{m-1} + \Phi_j \sum_{m=1}^{n_j} X_j^{m-1} + Y_j \sum_{m=1}^{n_j} \frac{X_j^{m-1} - 1}{X_j - 1} =$$

$$part_{j,0} \left(1 + Z_j \frac{X_j^{n_j} - 1}{X_j - 1} \right) +$$

$$\Phi_j \frac{X_j^{n_j} - 1}{X_j - 1} + \frac{Y_j}{X_j - 1} \left(\frac{X_j^{n_j} - 1}{X_j - 1} - n_j \right) = part_{j,0} V_j + W_j \quad (30)$$

where

$$V_j = \left(1 + Z_j \frac{X_j^{n_j} - 1}{X_j - 1} \right) \quad (31)$$

and

$$W_j = \Phi_j \frac{X_j^{n_j} - 1}{X_j - 1} + \frac{Y_j}{X_j - 1} \left(\frac{X_j^{n_j} - 1}{X_j - 1} - n_j \right) \quad (32)$$

Replacing equation (30) into equation (27) we get:

$$T_j = part_{j,0} \left(lsMK + l_P sMK + p_{j,a} NMK + l_P sNK + lNKsV_j \right) +$$
$$e_{j,a} + lsNM + 3b + lNKsW_j =$$
$$part_{j,0} K \left(p_{j,a} NM + l_P s(N+M) + ls \left(M + NV_j \right) \right) +$$
$$e_{j,a} + lsNM + 3b + lNKsW_j =$$
$$part_{j,0} Q_j + R_j + lsNM + 3b \quad (33)$$

where

$$Q_j = K \left(p_{j,a} NM + l_P s(N+M) + ls \left(M + NV_j \right) \right) \quad (34)$$

and

$$R_j = e_{j,a} + lNKsW_j \quad (35)$$

Optimality (the best possible execution time using the given hardware) dictates that the execution times of any pair of MN machines j and q are identical, hence:

$$T_j = T_q \Rightarrow part_{j,0} Q_j + R_j = part_{q,0} Q_q + R_q \Rightarrow$$

$$part_{j,0} = part_{q,0} \frac{Q_q}{Q_j} + \frac{R_q - R_j}{Q_j} \quad (36)$$

We can then combine equation (36) with the normalization equation to yield a closed form solution for the partitioning problem on a platform made-up of E MN nodes:

$$\sum_{j=0}^{E-1} \sum_{i=0}^{n_j} part\, j, i = 1 \overset{Eq.30}{\Rightarrow} \sum_{j=0}^{E-1} \left(part_{j,0} V_j + W_j \right) = 1 \Rightarrow$$

$$\sum_{j=0}^{E-1} \left(\left(part_{0,0} \frac{Q_0}{Q_j} + \frac{R_0 - R_j}{Q_j} \right) V_j + W_j \right) = 1 \Rightarrow$$

$$part_{0,0} = \frac{1 - \sum_{j=0}^{E-1} \left(V_j \frac{R_0 - R_j}{Q_j} + W_j \right)}{Q_0 \sum_{j=0}^{E-1} \frac{V_j}{Q_j}} \quad (37)$$

The pre-calculation of the X_j, Y_j, Z_j, Φ_j, V_j, W_j, Q_j, and R_j constants requires $\Theta(E)$ time and space. Subsequently, the calculation of the optimum partitioning requires the use of equation (37), equation (36) for $j \in [0, E-1]$ and equation (26) for $j \in [0, E-1]$ and $i \in [0, n_j - 1]$, in that order, for an overall time complexity of $\Theta(\sum_{j=0}^{E-1} n_j)$, i.e. linear with respect to the total number of cores in the system.

Chapter 5: Experiment Set-up and Measurement of Parameters

This chapter contains the hardware and software set-up followed by a description of the methodologies used to measure the parameters.

5.1. Hardware and Software Set-up

The proposed hardware set up is composed of highly heterogeneous computing cluster of three machines connected through an Ethernet 100 Mbps network. One machine will be the master node while the other two machines have high end dual GPU configuration. The detailed specifications of the machines are provided in Table 6.

Table 6: Hardware Specifications

	Dune-770	Kingpenguin	Dune-970
CPU	Core(TM) 2 Quad CPU Q8200 @ 2.33 GHz	Intel(R) Xeon(R) CPU E5-2640 @ 2.50GHz	Intel(R) Core(TM) i7-4820K CPU @ 3.70GHz
No. of CPU Cores	4	12	4
No. of Threads/Core	2	2	2
RAM	32 GB	64 GB	32 GB
No. of GPUs	2	-	1
GPU Version	GeForce GTX 770 & Quadro 5000	-	GeForce GTX 970
No. of GPU Cores	352	-	1536
GPU RAM	2559 MB	-	2048 MB
Compute capability	2.0 & 3.0	-	5.2

The software environment is the same for all the test beds in Table 7 and is mentioned below:

- Operating System: Kubuntu 15.04 (64 bit)
- Compiler: GCC 4.9.2, 64 bit
- Qt Version: 5.4.1
- CUDA Driver Version / Runtime Version 7.5 / 7.5.17
- OpenMPI 1.6.5
- cBLAS ATLAS Version and openBLAS Version 0.2.16.dev

5.2. Measurement of Parameters

The size of matrices A and B (floating point matrices) involved in the matrix product operation is determined by the values of the N, M and K variables. The product of N, M and K denoted by NMK is a good metric of the load as it indicates the number of floating point multiplications required to perform the matrix-matrix multiplication. Consequently, throughout this thesis, this size of data involved will be refereed to and plotted in terms of NMK.

In addition, the sizes of matrices used for testing this study are aimed to be as big as possible. The suggested hybrid technique is designed for huge matrices with matrix $B(M \times K)$ bigger than $A(N \times M)$. Matrix A is communicated as a whole to all nodes while matrix B is partitioned. To satisfy the above condition, the values of M and N is fixed to 10,000 while K ranges from 10,000 till the biggest possible number that allows A and B to fit in memory.

5.2.1. Measurement of CPU processing speed and latency. To measure the computation speed of the CPU, two random matrices A and B are generated with elements of type float in the range 1 to 10. As the model divides matrix B by columns, the matrices are assumed to be stored column-wise. Since C++ stores matrices row-wise, a transpose function is called on matrices A and B before passing them as arguments to the cblas_sgemm function. The test is repeated for 10 times and the time is accumulated after each iteration. The average time is taken by dividing the total time by 10; consequently, the resulting time is the CPU processing time of that particular NMK values.

The previous test is executed using a script that generates different NMK values. The average CPU time for each NMK is plotted in a graph in which the Y-axis represents time and the X-axis represents the NMK values, as shown in Figure 4. The slope of the line is the p_j while the intercept is e_j.

5.2.2. Measurement of GPU processing speed and latency. The matrixMulCUBLAS program that is shipped with the CUDA SDK samples is used to measure p_{ja}. A small adjustment was performed on the matrixMulCUBLAS source code, so that 10

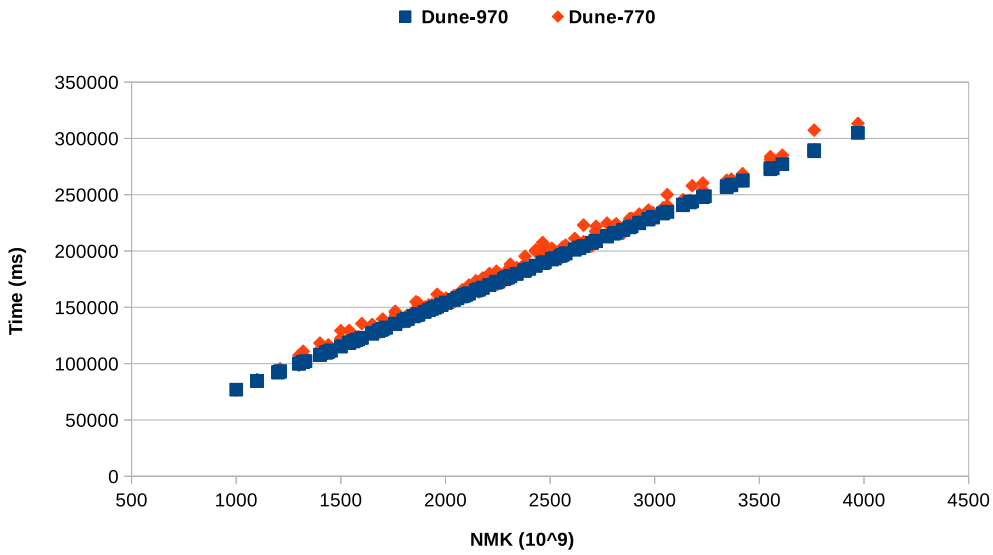

Figure 4: CPU processing time for different MNK values on Dune-970 and Dune-770. p_j is the slope and e_j is the intercept.

tests were conducted instead. e_{ja} is the sum of two variables; the first variable is the intercept resulting from plotting the matrix multiplication time on GPU versus NMK as shown in Figure 5, the second variable is the GPU initialization delay, mainly when the first cudaMalloc is called in the program, this is calculated manually by executing the CUDA code and recording the time elapsed while CUDA initialization functions are executing.

5.2.3. Measurement of GPU PCIe bus speed. cudaMemcpy is called several times using different array sizes of type float. The average values for Dune-970 and Dune-770 respectively, are shown in Figures 6 and 7. In both cases the slope of the least-squares line is used as the l_p.

5.2.4. Measurement of communication link speed and latency. A simple ping-pong program using MPI is used to measure the network speed. l is the slope of the line shown in Figure 8 while b is the intercept. Measuring this value is repeated 10 times and then the average value is computed.

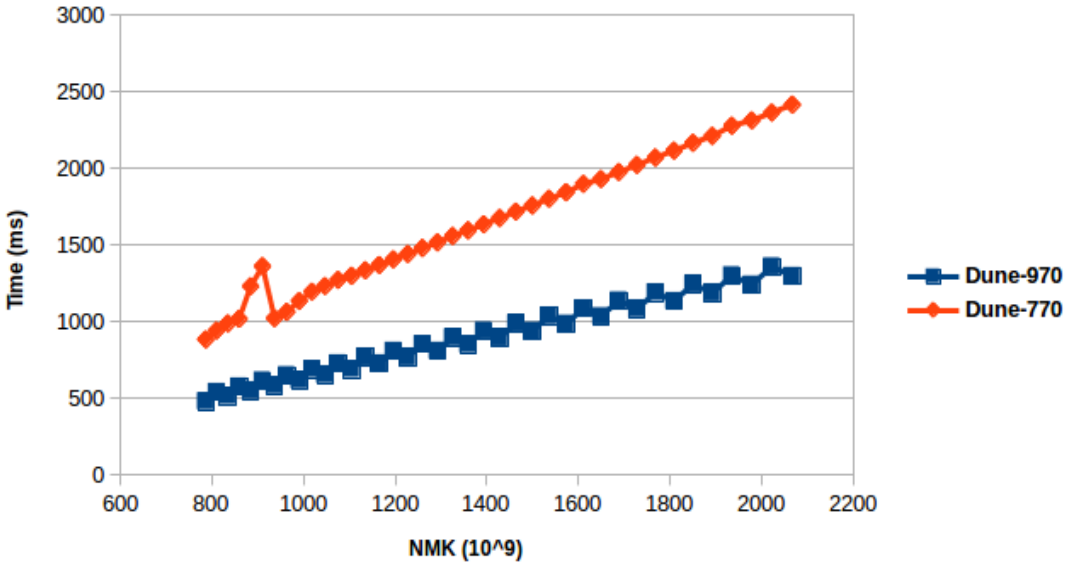

Figure 5: GPU processing time for different NMK values on Dune-770 and Dune-970. p_{ja} is the slope.

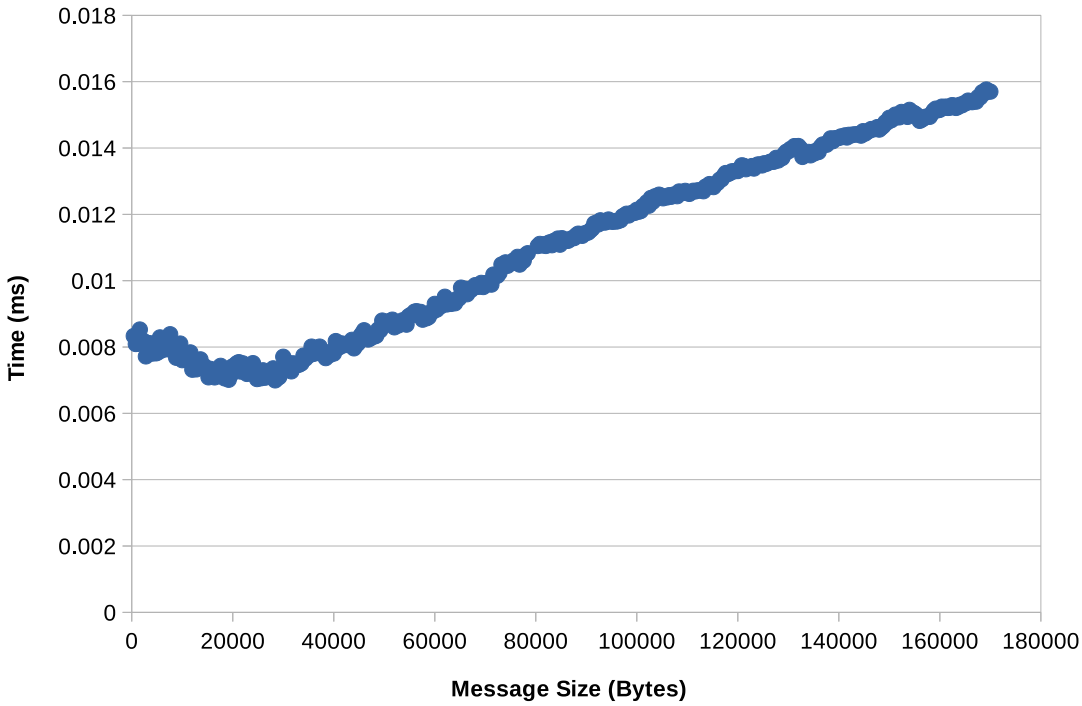

Figure 6: GPU PCIe bus speed on Dune-970. l_p is the slope.

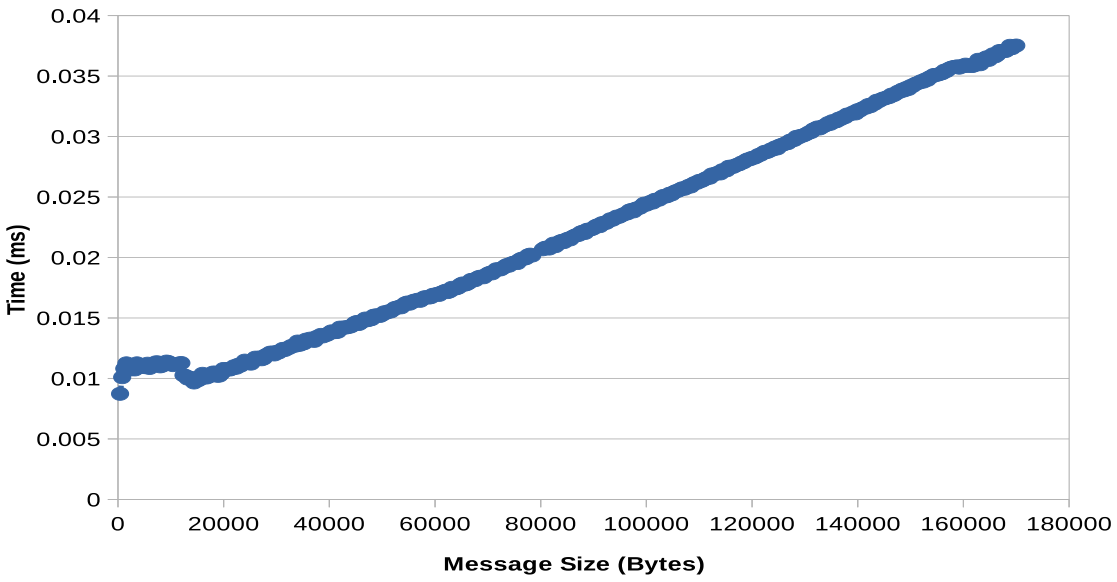

Figure 7: GPU PCIe bus speed on Dune-770. l_p is the slope.

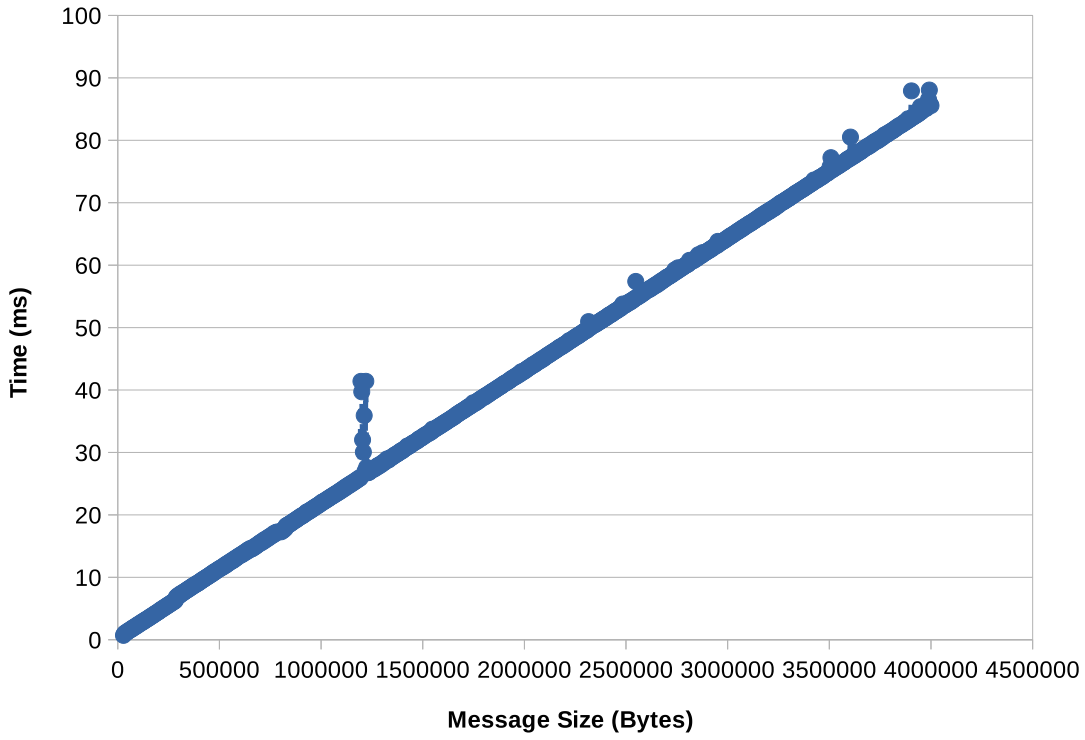

Figure 8: Graph showing the communication speed using a ping-pong program between Dune-970 and Kingpenguin. l is the slope and b is the intercept.

Chapter 6: Implementation and Profiling

Consider a small network of three processors, in which node 0 is the root, while nodes 1 and 2 are the workers. The worker nodes will read their properties (p_j, p_{ja}, e_j, etc.) from a properties file; these are the parameters required for calculating parts (mentioned in Section 4.2) and communicate these properties to the root node as shown in steps 2, 3, 7, 8, and 9 in Figure 9. Meanwhile the root reads matrix A from a file and broadcasts it to workers (steps 1, 4, 5, and 6). Afterwards, the root node will calculate the partitions using the closed form solution and use a collective MPI call (MPI_Scatter) to forward the parts in parallel to the corresponding nodes (steps 10 to 14).

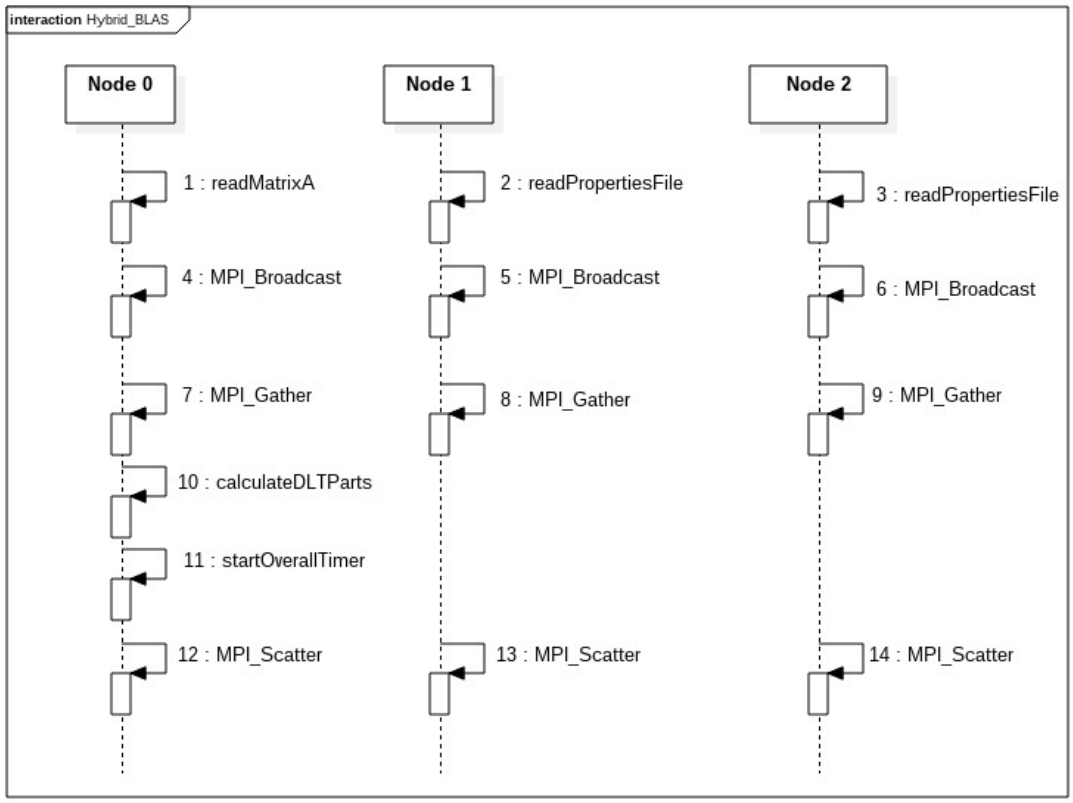

Figure 9: Sequence diagram of master-worker communication in the implementation of proposed matrix-matrix multiplication.

When each node receives its partition from the root node, it starts creating the threads (one thread for each processor). The node reads part of matrix B and then forwards matrix A as well as the columns of matrix B assigned to this processor to each thread sequentially as shown in steps 5 to 13 in Figure 10. Each thread will then call

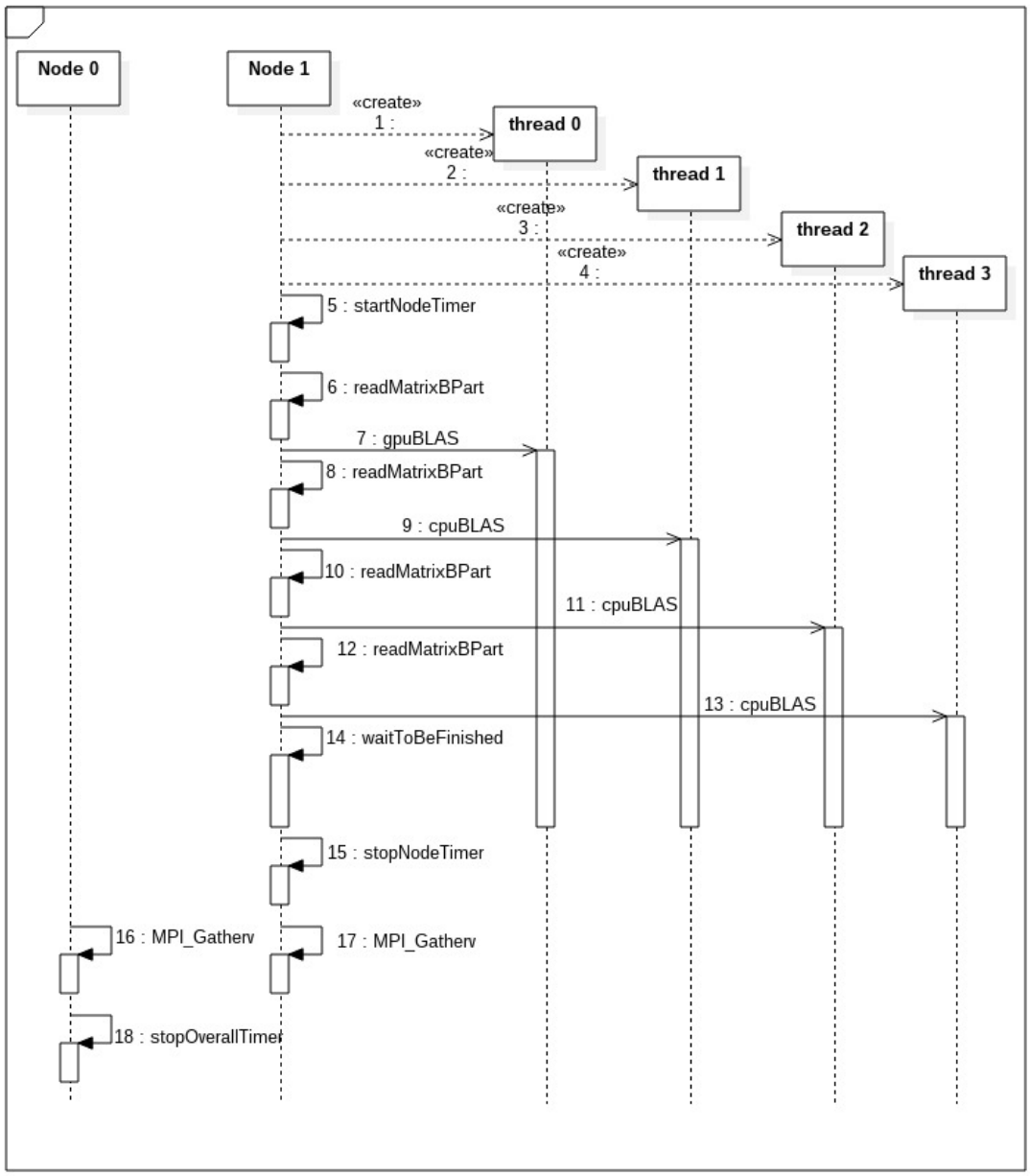

Figure 10: Sequence diagram showing internode communication in the implementation of matrix-matrix multiplication.

the corresponding BLAS operation (cblas_sgemm for the CPU and cublasSgemm for the GPU). Afterwards, the worker node will wait for all threads to finish computation and then forwards the result (its part of matrix *C*) to the root through the collective MPI call MPI_Gatherv (steps 16 and 17).

There are two timings measured in this implementation to assess performance. The first is the overall time for matrix-matrix multiplication calculated in the root node

and the second is the processing time spent by the node to compute the load named as NodeTimer (steps 5 and 15 in Figure 10).

In addition, VampirTrace is used to monitor the performance of the proposed solution. VampirTrace is an open source library used to instrument and trace parallel software applications. After a successful run for the application, vampirTrace stores the collected data in an OTF file. This OTF file is visualized by an open source software called ViTE. More information about vampirTrace and ViTE can be found in [44] and [45], respectively.

Chapter 7: Results and Discussion

All the matrices used for testing the hybrid methodology are generated using a random number generator that generates floating point numbers between 1 and 10 and stores them in a binary file. As this technique is designed for huge matrices with matrix B bigger than A, the values of M N K must be as big as possible. The values of M and N will be fixed to 10,000 while K value ranges from 10,000 to 41,000 (the biggest possible value that allows A and B to fit in memory).

7.1. Hybrid Approach on Dune-970

The parameters of Dune-970 are listed in Table 7. The time was measured 10 times and the average is calculated for both hybrid approach and the GPU only (using cuBLAS to process all load) method as shown in Figure 11.

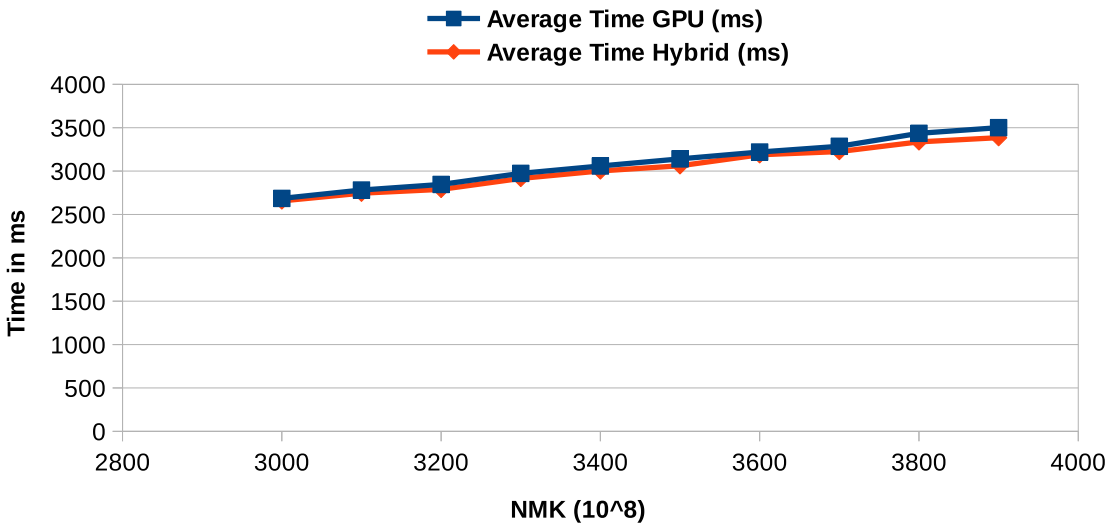

Figure 11: Showing the GPU only time and the hybrid time on Dune-970

In Table 7, the value of p_{ja} is much smaller than p_j, which indicates that GPU performance exceeds CPU performance in this matrix product by a big factor (two orders of magnitude). Consequently, assigning a big portion of the load to the GPU in the hybrid approach is sensible. This can be seen in Table 8 in which the percentage of load assigned to GPU is never below 98% in all N M K values. This explains the

small speed-up achieved when using the hybrid approach compared to the GPU only one (Figure 11 and 12).

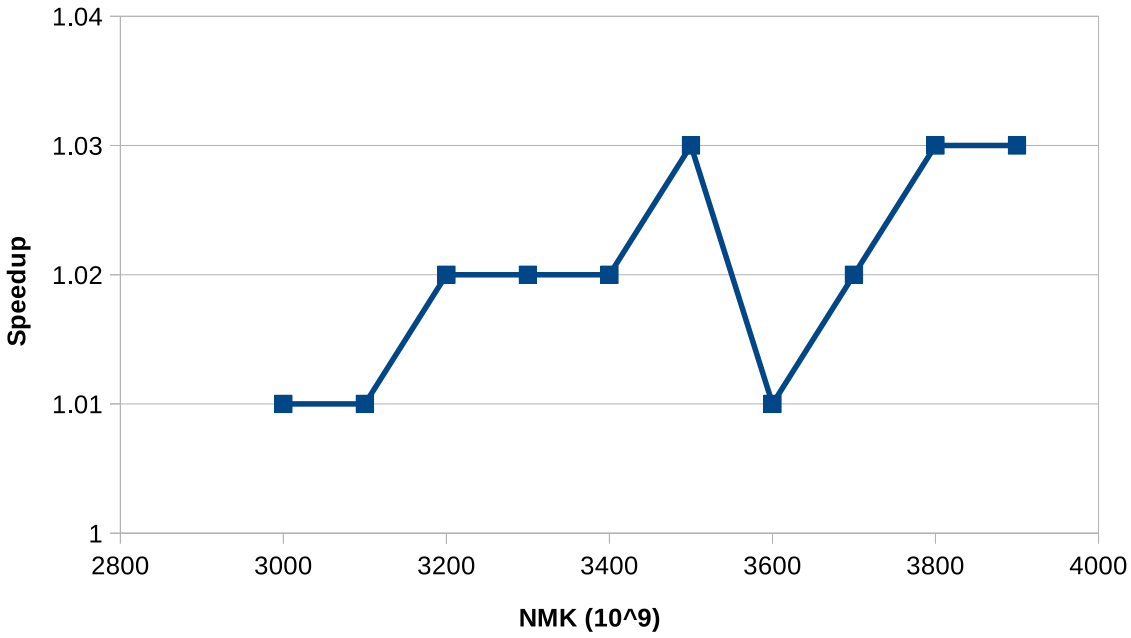

Figure 12: Speedup achieved by hybrid approach on Dune-970

Table 7: Dune-970 properties

p_j	7.677E-008 msec/byte
e_j	132.16 msec
p_{ja}	6.4932E-010 msec/byte
e_{ja}	175.5 msec
l_p	5.332E-008 msec/byte

7.2. Hybrid Approach on Dune-770

The proposed hybrid approach was also tested on Dune-770. The execution time was measured 10 times and the average was calculated for the hybrid, the GPU only, CPU cBLAS, and CPU openBLAS methods, as shown in Figures 13 and 14. The GPU only, as well as hybrid methodologies, are much faster than cBLAS, and openBLAS ones. In some cases, the hybrid approach performance slightly exceeds the GPU only method. Similar to Dune-970, the difference between CPU and GPU processing powers

Table 8: Summary of the results collected from Dune-970

NMK	Average Time GPU only (in ms)	Average Time Hybrid (in ms)	Fraction of load assigned to GPU in Hybrid approach
10K * 10K * 30K	2685.48	2656.63	0.981259
10 K * 10K * 31K	2781.97	2745.08	0.981271
10K * 10K * 32K	2846.13	2788.96	0.981283
10K * 10K * 33K	2973.84	2915.51	0.981293
10K * 10K * 34K	3058.99	3001.59	0.981303
10K * 10K * 35K	3140.77	3061.46	0.981313
10K * 10K * 36K	3219.61	3187.68	0.981322
10K * 10K * 37K	3285.86	3225.39	0.98133
10K * 10K * 38K	3435.22	3337.11	0.981338
10K * 10K * 39K	3500.67	3385.43	0.981346

is very big as shown in Table 10. Consequently, the minimum load in hybrid approach assigned to GPU never falls below 97% (see Table 10). As a result, there is a small performance difference between the GPU only approach and the hybrid one on Dune-770.

In addition, monitoring Dune-770 performance using VampirTrace reveals the success of the hybrid method load distribution strategy, as the generated traces show the three CPUs and the GPU finish execution the same time.

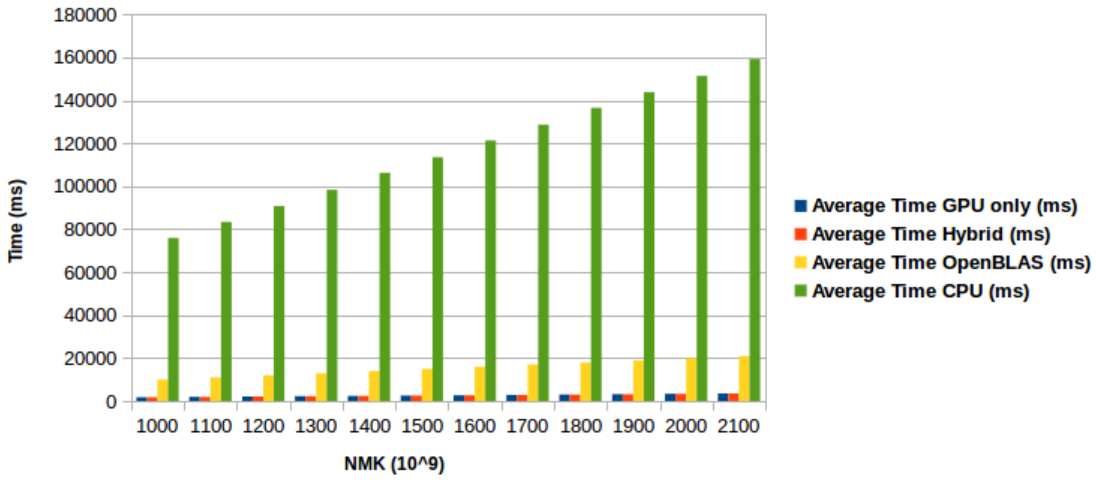

Figure 13: Comparative results of GPU only, hybrid, cBLAS and openBLAS methods on Dune-770

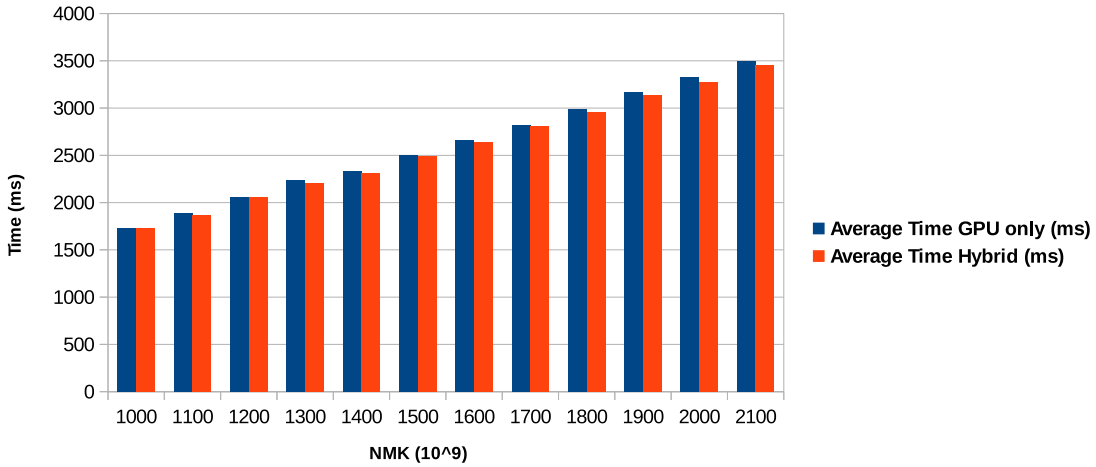

Figure 14: Comparative results of GPU only and hybrid methods on Dune-770

Table 9: Dune-770 properties

p_j	7.8677E-008 msec/byte
e_j	297.5628 msec
p_{ja}	1.10844E-009 msec/byte
e_{ja}	100 msec
l_p	6.8828E-008 msec/byte

Table 10: Summary of the results collected from Dune-770

NMK	Average Time GPU only (ms)	Average Time Hybrid (ms)	Average Time CPU only cBLAS (ms)	Average Time CPU openBLAS (ms)	Fraction of load assigned to GPU in Hybrid Approach
10K * 10K * 10K	1725.44	1724.94	75916.61	10025.19	0.975471
10 K * 10K * 11K	1885.51	1863.49	83335.83	10968.35	0.975149
10K * 10K * 12K	2061.02	2057.19	90776.37	11968.44	0.974881
10K * 10K * 13K	2234.02	2204.55	98340.89	12892.53	0.974654
10K * 10K * 14K	2333.54	2312.81	106241.2	13944.4	0.974459
10K * 10K * 15K	2502.25	2485.42	113462.4	14957.68	0.974291
10K * 10K * 16K	2663.02	2636.52	121272.1	15896.15	0.974143
10K * 10K * 17K	2822.33	2802.94	128682.4	16925.58	0.974013
10K * 10K * 18K	2989.93	2956.32	136505.3	17889.73	0.973897
10K * 10K * 19K	3162.42	3134.75	143799.3	18831.22	0.973794
10K * 10K * 20K	3328.97	3278.58	151380.1	19869.82	0.973701
10K * 10K * 21K	3499.38	3454.31	159144.5	20866.12	0.973616

7.3. Hybrid Approach on Multiple Nodes

Testing the hybrid approach on multiple nodes was conducted on Kingpenguin, Dune-970 and Dune-770. Kingpenguin served as the root node, hence it did not take part in the computation. The load was divided between Dune-970 and Dune-770, and the processing time was computed for each. On the other hand, the overall timing was calculated by Kingpenguin; this overall time includes matrix communication (except for matrix A), as well as computational time. Due to slow communication, the reduction in processing time resulting from load division was masked by the communication time; however, the loads on the two machines were balanced and they finished computation almost at the same time as shown in Table 11.

In addition, the time calculated using equation (27) in Section 4.2.1.1 greatly reflects the overall time calculated by Kingpenguin for different NMK values. The graph plotted in Figure 15 shows the success of the DLT equation in predicting the execution time for NMK values ranging from 1000×10^9 to 4100×10^9. The time calculated using equation (22) (excluding matrix A communication) is considered the processing time predicted by the DLT theory. The overall time calculated by Kingpenguin is depicted as Measured.

Table 11: Summary of the results collected from multiple node test

NMK (x 10^9)	Average Processing Time Node 1 (Dune-770) in ms	Average Processing Time Node 2 (Dune-970) in ms	Overall Time in ms
1000	842.77	720.49	35703.94
1100	922.97	767.31	39051.97
1200	1011.54	805.59	42692.15
1300	1100.34	838.15	46393.11
1400	1180.03	885.16	49774.11
1500	1268.08	943.07	53669.52
1600	1340.98	983.85	57333.05
1700	1427.32	1028.46	60700.7
1800	1505.95	1068.64	60700.7
1900	1588.33	1138.89	64210.48
2000	1679.79	1157.46	67627.1
2100	1754.38	1214.2	71034.28

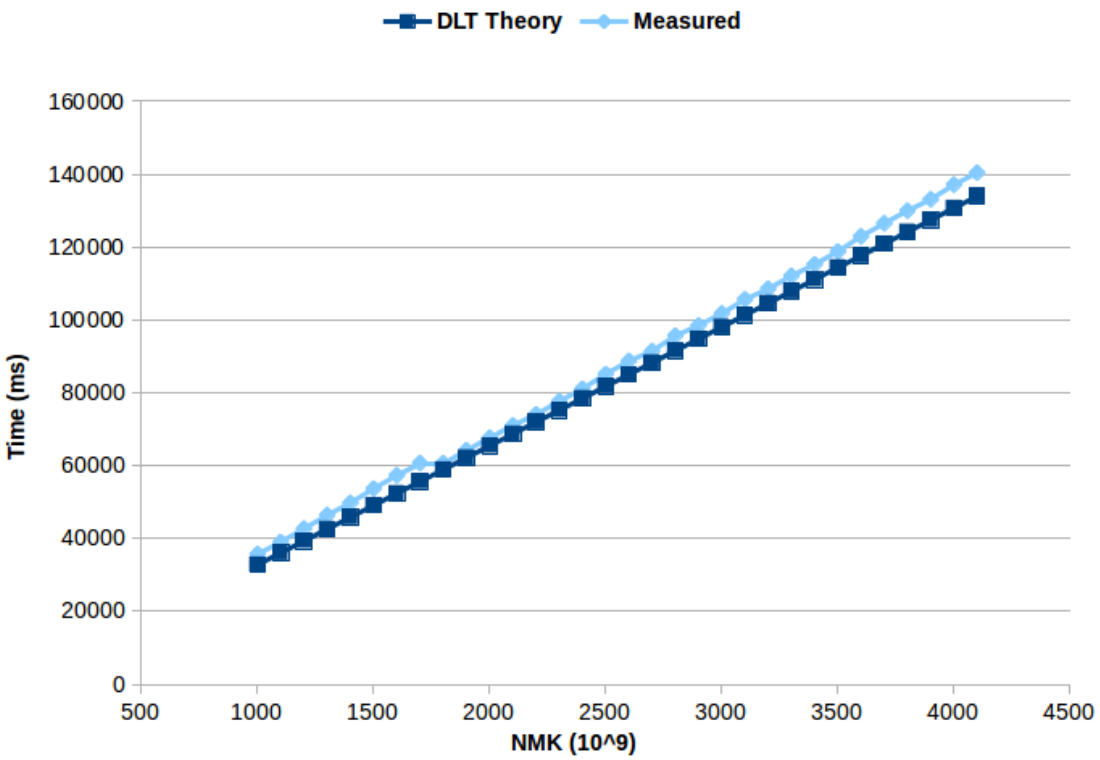

Figure 15: Graph showing expected execution time calculated using DLT equation (shown as DLT Theory) and actual execution time (shown as Measured) in multiple nodes experiment.

Chapter 8: Conclusion

Hybrid CPU-GPU systems recently attracted the attention of the parallel software community aiming at further performance enhancement. The biggest challenge facing hybrid computation success is load balancing. This study suggests a load partitioning approach to improve efficiency of matrix-matrix multiplication on a distributed network composed of heterogeneous nodes, using the DLT methodology. The proposed technique efficiently handles the inter-node and intra-node load balancing to reduce the overall execution time.

The provided solution used DLT analysis to acquire the closed form solution. The parameters of the available hardware were accurately measured followed by running several experiments on both single and multiple nodes.

The results of the hybrid approach on a single node showed that the acquired load balance between the GPU and available cores was achieved. This was also confirmed by the VampirTrace profiler. In addition, the hybrid approach showed significant speed-up compared to cBLAS, and openBLAS methods. However, there is a small difference between the hybrid and cuBLAS approaches as the portion assigned to the GPU formed about 97% to 98% of the load. This result is justified by the big difference in performance between the GPU and CPU in the used hardware. In the multiple nodes case, the DLT equations succeed in predicting the real execution time. However, the overall time of the matrix product on multiple nodes was inferior to a single GPU time due to slow communication.

Future work may include dividing the load between GPUs on a single node in case the node is equipped with more than one GPU. The above study suggests that dividing the load between GPUs on a single node will achieve speed-up for the following reasosns:

- No slow communication involved, as the experiment is done an a single node.
- The difference in performance between the GPUs will allow them to share evenly in load processing and thus further speed-up could be achieved.

References

[1] S. Suresh, R. Cui, K. Hyoung Joong *et al.*, "Scheduling Second-Order Computational Load in Master-Slave Paradigm," *IEEE Transactions on Aerospace & Electronic Systems*, vol. 48, no. 1, pp. 780–793, 2012.

[2] Z. Wang, Q. Chen, L. Zheng *et al.*, "CPU+GPU Scheduling With Asymptotic Profiling," *Parallel Computing*, vol. 40, no. 2, pp. 107–115, 2013.

[3] G. Barlas, *Multicore and GPU Programming an Integrated Approach*. USA: MK, 2015, pp. 575–628.

[4] O. Beaumont, H. Larcheveque, and L. Marchal, "Non Linear Divisible Loads: There is No Free Lunch," in *IEEE 27th International Symposium Parallel & Distributed Processing (IPDPS)*, 2013, pp. 863–873.

[5] A. Ilic and L. Sousa, "On Realistic Divisible Load Scheduling in Highly Heterogeneous Distributed Systems," in *2012 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2012, Conference Proceedings, pp. 426–433.

[6] T. G. Robertazzi, "Ten Reasons to Use Divisible Load Theory," *Computer*, vol. 36, no. 5, pp. 63–68, 2003.

[7] A. Shokripour and M. Othman, "Survey on Divisible Load Theory," in *2009 International Association of Computer Science and Information Technology - Spring Conference*, 2009, Conference Proceedings, pp. 9–13.

[8] S. Ghanbari and M. Othman, "Comprehensive Review on Divisible Load Theory: Concepts, Strategies, and Approaches," *Mathematical Problems in Engineering*, vol. 2014, pp. 1–13, 2014.

[9] G. Tao, L. Qun, Y. Yulu *et al.*, "Scheduling Divisible Loads from Multiple Input Sources in MapReduce," in *2013 IEEE 16th International Conference on Computational Science and Engineering (CSE)*, 2013, Conference Proceedings, pp. 1263–1270.

[10] What is GPU Computing. Internet. [Online]. Available: www.nvidia.co.uk/object/gpu-computing-uk.html [August 2, 2016].

[11] C. Rosas, A. Sikora, J. Jorba *et al.*, "Improving Performance on Data-Intensive Applications Using a Load Balancing Methodology Based on Divisible Load Theory," *International Journal of Parallel Programming*, vol. 42, no. 1, pp. 94–118, 2014.

[12] CUDA C Programming Guide. Internet. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf [June 10, 2016].

[13] A. J. Park and K. S. Perumalla, "Efficient Heterogeneous Execution on Large Multicore and Accelerator Platforms: Case Study Using a Block Tridiagonal Solver," *Journal of Parallel and Distributed Computing*, vol. 73, no. 12, pp. 1578–1591, 2013.

[14] G. Seber, *A Matrix Handbook for Statisticians*. Wiley, 2008.

[15] S. P. Hirshman, K. S. Perumalla, V. E. Lynch *et al.*, "BCYCLIC: A parallel Block Tridiagonal Matrix Cyclic Solver," *Journal of Computational Physics*, vol. 229, no. 18, pp. 6392–6404, 2010.

[16] V. T. Ravi, M. Becchi, W. Jiang *et al.*, "Scheduling Concurrent Applications on a Cluster of CPUGPU Nodes," *Future Generation Computer Systems*, vol. 29, no. 8, pp. 2262–2271, 2013.

[17] E. Zhu, R. Ma, Y. Hou *et al.*, "Two-phase Execution of Binary Applications on CPU/GPU Machines," *Computers & Electrical Engineering*, vol. 40, no. 5, pp. 1567–1579, 2014.

[18] A. Lastovetsky and R. Reddy, "Data Partitioning for Multiprocessors with Memory Heterogeneity and Memory Constraints," *Scientific Programming*, vol. 13, no. 2, pp. 93–112, 2005.

[19] G. Barlas, A. Hassan, and Y. Al Jundi, "An Analytical Approach to the Design of Parallel Block Cipher Encryption/Decryption: A CPU/GPU Case Study," in *2011 19th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2011, Conference Proceedings, pp. 247–251.

[20] G. Barlas, "Cluster-based Optimized Parallel Video Transcoding," *Parallel Computing*, vol. 38, no. 45, pp. 226–244, 2012.

[21] L. Ping, B. Veeravalli, and A. A. Kassim, "," *IEEE Transactions on Circuits & Systems for Video Technology*, vol. 15, no. 9, pp. 1098–1112, 2005.

[22] S. Momcilovic, A. Ilic, N. Roma *et al.*, "Dynamic Load Balancing for Real-Time Video Encoding on Heterogeneous CPU+GPU Systems," *IEEE Transactions on Multimedia*, vol. 16, no. 1, pp. 108–121, 2014.

[23] S. Suresh, H. Huang, and H. J. Kim, "Scheduling in Compute Cloud with Multiple Data Banks Using Divisible Load Paradigm," *IEEE Transactions on Aerospace & Electronic Systems*, vol. 51, no. 2, pp. 1288–1297, 2015.

[24] C.-k. Lee and M. Hamdi, "Parallel Image Processing Applications on a Network of Workstations," *Parallel Computing*, vol. 21, no. 1, pp. 137–160, 1995.

[25] V. Bharadwaj, X. Li, and C. C. Ko, "Efficient Partitioning and Scheduling of Computer Vision and Image Processing Data on Bus Networks Using Divisible Load Analysis," *Image and Vision Computing*, vol. 18, no. 11, pp. 919–938, 2000.

[26] G. Barlas, "An Analytical Approach to Optimizing Parallel Image Registration/Retrieval," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 8, pp. 1074–1088, 2010.

[27] B. Veeravalli and S. Ranganath, "Theoretical and Experimental Study on Large Size Image Processing Applications Using Divisible Load Paradigm on Distributed Bus Networks," *Image and Vision Computing*, vol. 20, no. 13, pp. 917–935, 2002.

[28] D. H. P. Low, B. Veeravalli, and D. A. Bader, "On the Design of High-Performance Algorithms for Aligning Multiple Protein Sequences on Mesh-Based Multiprocessor Architectures," *Journal of Parallel and Distributed Computing*, vol. 67, no. 9, pp. 1007–1017, 2007.

[29] W. H. Min and B. Veeravalli, "Aligning Biological Sequences on Distributed Bus Networks: a Divisible Load Scheduling Approach," *IEEE Transactions on Information Technology in Biomedicine*, vol. 9, no. 4, pp. 489–501, 2005.

[30] S. Haiyan, W. Wanliang, K. Ngai Ming *et al.*, "Adaptive Indexed Divisible Load Theory for Wireless Sensor Network Workload Allocation," *International Journal of Distributed Sensor Networks*, vol. 2013, pp. 1–18, 2013.

[31] M. Moges and T. G. Robertazzi, "Wireless Sensor Networks: Scheduling for Measurement and Data Reporting," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 42, no. 1, pp. 327–340, 2006.

[32] H. Shi, W. Wang, and N. Kwok, "Energy Dependent Divisible Load Theory for Wireless Sensor Network Workload Allocation," *Mathematical Problems in Engineering*, vol. 2012, pp. 1–16, 2012.

[33] H. Shi, N. Kwok, W. Wang *et al.*, "Divisible Load Theory Based Active-Sleep Workload Assignment Schemes for Wireless Sensor Networks," *International Journal of Distributed Sensor Networks*, vol. 2014, pp. 1–18, 2014.

[34] P. Thysebaert, M. De Leenheer, B. Volckaert *et al.*, "Scalable Dimensioning of Resilient Lambda Grids," *Future Generation Computer Systems*, vol. 24, no. 6, pp. 549–560, 2008.

[35] M. Abdullah, M. Othman, H. Ibrahim *et al.*, "Optimal Workload Allocation Model for Scheduling Divisible Data Grid Applications," *Future Generation Computer Systems*, vol. 26, no. 7, pp. 971–978, 2010.

[36] S. K. Chan, V. Bharadwaj, and D. Ghose, "Large MatrixVector Products on Distributed Bus Networks with Communication Delays Using the Divisible Load Paradigm: Performance Analysis and Simulation," *Mathematics and Computers in Simulation*, vol. 58, no. 1, pp. 71–92, 2001.

[37] Basic Linear Algebra Subprograms (BLAS). Internet. [Online]. Available: www.netlib.org/blas/#_presentation [June 7, 2016].

[38] CuBLAS Library. Internet. [Online]. Available: www.ecse.rpi.edu/homepages/wrf/wiki/ParallelComputingSpring2014/nvidia/cuda6doc/pdf/CUBLAS_Library.pdf [June 2, 2016].

[39] AMD Core Math Library. Internet. [Online]. Available: amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/acml.pdf [June 11, 2016].

[40] Intel Math Kernel Library. Internet. [Online]. Available: software.intel.com/en-us/intel-mkl [June 11, 2016].

[41] M. Shevtsav. OpenCL: Advantages of the Heterogeneous Approach. Internet. [Online]. Available: software.intel.com/en-us/articles/opencl-the-advantages-of-heterogeneous-approach [July 21, 2016].

[42] H. Brunst, M. S. Mller, W. E. Nagel *et al.*, *HiFlow3: A Hardware-Aware Parallel Finite Element Package*. Springer Berlin Heidelberg, 2012, pp. 139–151.

[43] S. Chuprat, S. Salleh, and S. Goddard, "Real-Time Divisible Load Theory: A Perspective," in *2009 International Conference on Parallel Processing Workshops (ICPPW)*, 2009, Conference Proceedings, pp. 6–10.

[44] VampirTrace-User Manual. Internet. [Online]. Available: tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/projekte/vampirtrace/accelerator/dateien/VampirTraceManual/VampirTrace%205.14.3-gpu1-user-manual.html [June 21, 2016].

[45] ViTE-User Manual. Internet. [Online]. Available: vite.gforge.inria.fr/documentation.php [June 16, 2016].

Vita

Lamees Elhiny graduated from the American University in Cairo, Computer Science Major, Fall 2005 with high honors. Immediately after graduation, she joined IBM Egypt where she worked in several E-Government projects using Java, JSP, JavaScript, and other web development technologies. She started as a subcontractor but later succeeded to be an IBMer by 2010. She moved to the United Arab Emirates in 2013, and started her master degree in Computer Engineering, where she gained in depth knowledge of C++, parallel processing, and internet computing.